Aspects, Dependencies and Interactions
Report on the WS ADI at ECOOP 2007

Frans Sanen[1], Ruzanna Chitchyan[2], Lodewijk Bergmans[3],
Johan Fabry[4], Mario Sudholt[5], and Katharina Mehner[6]

[1]K.U.Leuven, Leuven, Belgium, frans.sanen (at) cs.kuleuven.be
[2]Lancaster University, Lancaster, UK, rouza (at) comp.lancs.ac.uk
[3]University of Twente, Enschede, The Netherlands, L.M.J.Bergmans (at) ewi.utwente.nl
[4]INRIA Futurs, LIFL, France, johan.fabry (at) lifl.fr
[5]Ecole des Mines de Nantes, Nantes, France, Mario.Sudholt (at) emn.fr
[6]Siemens, Germany, Katharina.Mehner (at) siemens.com

Abstract. The topics on aspects, dependencies and interactions are among the key remaining challenges to be tackled by the Aspect-Oriented Software Development (AOSD) community to enable a wide adoption of AOSD technology. This second workshop, organized and supported by the AOSD-Europe project, aimed to continue the wide discussion on aspects, dependencies and interactions started at ADI 2006.

Keywords. Aspects, dependencies, interactions

1 Introduction

Aspects are crosscutting concerns that exist throughout the software development life cycle - from requirements through to implementation. While crosscutting other concerns, aspects often exert broad influences on these concerns, e.g., by modifying their semantics, structure or behaviour. These dependencies between aspectual and non-aspectual elements may lead to either desirable or (more often) unwanted and unexpected interactions. The goal of this second workshop was to continue the wide discussion on aspects, dependencies and interactions started at ADI 2006, thus investigating the problems of aspects, dependencies and interactions and handling them at all levels:

- starting from the early development stages (i.e., requirements, architecture, and design), looking into dependencies between requirements (e.g., positive/negative contributions between aspectual goals) and interactions caused by aspects (e.g., quality attributes) in requirements, architecture, and design;
- analyzing these dependencies and interactions both through modeling and formal analysis;
- considering language design issues which help to handle such dependencies and interactions (e.g., 'declare precedence' mechanism of AspectJ);
- studying such interactions in applications.

In the rest of this workshop report, we present the main topics that were discussed at the workshop, including a comparative overview of the main topics of the accepted papers, a summary of the keynote speech by Gary T. Leavens on "Concerning efficient reasoning in AspectJ-like languages, a summing-up of the debates hold in the different discussion breakout groups and a synthesis of the panel chaired by Awais Rashid on "Does AO equal quantification and obliviousness?"

2 Accepted Papers

Papers accepted to the workshop covered a broad spectrum of problems related to aspects, dependencies and interactions. We have clustered these papers into three sets, with each set briefly summarized below.

2.1 Requirements, Analysis and Design

This set of papers focuses mainly on the early stages of AOSD: requirements engineering, analysis and design modeling.

In [3], a method is proposed that supports the identification of functional requirements that crosscut other functional requirements. In addition, guidelines about how to generate derived or modified requirements are provided. The authors of the paper use actions as the primary means for identifying match-points between functional requirements. The authors propose to manually define a list of all actions that are directly used by each action, i.e., the implied actions. These implied actions are then used to check whether requirements crosscut each other. Modes or states of the different entities in the system are also important for determining whether requirements crosscut: requirements related to the same mode crosscut while requirements with mutually exclusive modes do not crosscut each other. Next to implied actions and modes, action modifiers are described to help to decide whether two requirements crosscut each other. The authors distinguish between three action modifiers: restrict, unconditional and none. If a requirement restricts the use of an action X, then all actions that imply action X are also restricted. A similar observation is made for requirements that can be used unconditionally.

In [19], the authors argue that currently it is difficult to verify whether a base model is correctly structured and if the weaving reflects the intention of a modeler. They propose a verification method for weaving in AspectM: an extensible aspect-oriented modeling language [20, 21]. The paper focuses on the verification of the base model. AspectM provides not only major join point mechanisms but also a mechanism called meta-model access protocol that allows a modeler to modify the meta-model, which is an extension of the UML meta-model. Prototype tool support for the reflective model editor and model weaver has been developed. The tool consists of a meta-model checker for verifying whether a base model conforms to the meta-model, a module structure checker for detecting the aspect interference and an assertion checker.

2.2 Language-level Problems

This set of papers looks at novel AO language concepts regarding aspect interaction management and issues of interaction between aspects written in different AO domain-specific languages.

In [4], novel concepts regarding aspect interaction management are defined. The paper proposes some extensions to the AspectJ [2] language for detecting unintended aspect interactions. These extensions are aspect and advice cardinality, and meta-aspects. The authors start off by providing a classification of seven different types of aspect interactions. Some fundamental causes of undesired interactions also are discussed. Next, aspect and advice cardinality is defined to represent the absolute and relative proportions of aspect use and advice weaving. Aspect cardinality is the measure

of the expected number of aspect bindings to an application while advice cardinality represents the expected number of advice weavings per aspect binding. It's the developer's responsibility to ensure that multiple weavings at the same join point behave coherently depending on a certain applicable execution order. Finally, meta-aspects are generic, abstract specifications of concrete aspects with a number of advantages. These concrete aspects usually can be derived automatically with all generic pointcut definitions being instantiated into specific, narrow-scoped expressions.

In [14], the authors focus on understanding interactions between foreign aspects, i.e., aspects written in different aspect domain-specific languages. They distinguish between two categories: co-advising and foreign advising. Co-advising is the application of multiple pieces of advice to the same join point while foreign advising captures the situation where an aspect also advises aspects written in languages other than the base. A classification and comparison of a set of composition approaches according to whether these resolve the interactions at the language level or at the program level is covered in the paper. In order to understand why resolving these interactions at the language level is fundamentally different than resolving them at the program level, the authors elaborate on both the Reflex [18] and Awesome [13] frameworks. The latter handles both foreign advising and co-advising interactions.

2.3 Contract-based Approaches

This set of papers addresses contract-based approaches for managing aspect interactions in an AO middleware or for controlling use of aspects without constraining the power of AOSD.

In [9], the authors aim to manage interaction issues in an aspect-oriented middleware platform by allowing interaction contracts to be specified which then are enforced at runtime. Explicitly specifying these contracts improves the management and control of such interactions. The work focuses on two broad categories of aspect interactions: conflicts (two aspects being incompatible) and dependencies (one aspect requiring another). The solution in the paper includes a component model with a well-defined interaction model that supports a variety of relationships. These relationships are specified using interaction contracts that are evaluated at runtime to ensure conflicts do not occur and dependencies are fulfilled. The interaction model is based on shared elements (such as a common join point, a component instance or the base application). It's possible to specify both basic (requires and provides) and advanced (conflict, precedence and resolution) interaction contracts. The approach has been validated by applying it to a series of interaction issues that occurred when implementing services for a flexible and customizable AO middleware platform, CustAOMWare.

In [16], an overview is given of approaches that address two important challenges for AOSD's mainstream adaption: the evolution paradox problem and the invasive nature of aspects. The evolution paradox encompasses the difficulties that arise when an application created using AOSD tries to evolve and is hampered by the fragile pointcut problem. Invasive aspects enable us to specify harmful advice that breaks encapsulation. As a consequence, aspects can invalidate some of the already existing desirable properties of a system resulting in, among others, security problems. Current approaches that solve or reduce one or both of these problems are categorized according to the means they use: guidelines, code-based, analysis, model-based and contract-based. Next, a solution to deal with these problems is sketched. The aim is not to

constrain the power of AOSD, but rather control aspect invasiveness and fit aspects to better support evolution.

3 Keynote Speech by Gary T. Leavens on "Concerning Efficient Reasoning in AspectJ-like Languages"

The work presented in this keynote was concerned with efficient forms of reasoning. The approach taken was based on static analysis of source code that is annotated with meta-information, to determine (non-)interference of the aspect with the base code. What was specified in this approach is object state and method preconditions, heap effects and control effects. Heap effects include postconditions such as changes to static variables. Control effects treat how the control flow of the method is changed. Specification is done by the developer through annotations of the source code. Some of these annotations are deducible, however no support for automatic deduction is provided. Implementation verification is performed by a conservative static analysis. The reasoner accumulates facts as the program is processed, and verifies these with regard to the given specifications.

The innovative part of this approach over existing reasoning with contracts approaches was that it has been effectively tailored towards aspects. This is because the classical subtyping relationships used in these approaches are not applicable to aspects. Around advice can be considered like an overriding method, but is often used to change the behavior in different ways than what an overriding method would do. For example, advice introduces a number of control effects, such as running the original method multiple times. The existing specification approaches cannot reflect this, and their verification steps are not designed for it.

Multiple possible and existing approaches for specification of advice were then discussed. The first approach was using the semantics directly, which is maximally expressive, but implies re-verification for all changes, and provides no abstraction. A second approach is considering functional advice, which has no heap or control effects, which does not affect reasoning over the base code. This advice cannot do anything however, and is therefore useless. Third, the concept of *harmless advice* [6] was discussed. Here no information flows from the advice to the base code, which has as benefits that no heap effects on the base occur. The downsides are that this does not address control effects, there is a loss of expressiveness and inference amongst advice is not addressed. A fourth approach is a refinement of behavioral subtyping, using an object-oriented analogy of around advices as overriding methods, and proceed as a super call. This allows the base code to be verified independent of the advice. This has a number of downsides such as that quantification is limited, and that much advice falls outside of this paradigm, as said above. In general the downsides are too important to make this feasible. A fifth approach is specifying at the language level which join points can be advised, as proposed by multiple authors. This poses no limits on expressive power, but has as downside, amongst others, that interference amongst advices is not considered.

The last approach discussed was reasoning about the level of specifications, written in the aspect, that are woven. The presenter expects that this is the direction that the community is taking. If successful, this has the benefits that it is at a more abstract level than code, and will allow changes in methods and advice without the need for re-verification. The downsides are that there is less expressiveness and that the weaving

of specifications is difficult and expensive. However, there are some optimizations that are possible, e.g., ignoring inapplicable advice and spectator advice, which do not affect the heap nor has control effects. A second form of optimization is via effect analysis: an advice heap interferes with base code if it writes a field that is read in the base code. This is efficient because it only needs to look at signatures, and furthermore the analysis can also apply to two pieces of advice.

The last part of the talk then gave an overview of concern domains. It follows the specification weaving approach above, by declaring concern domains, i.e., partitions of the heap, in which write effects are declared. A form of type and effect analysis is then used to detect potential interference. This has been proven to be sound for checking possible heap interference. A further benefit is that spectators can be ignored in the verification phase. Downsides are the cost incurred by manually declaring the effects of methods and advice, and a number of restrictions placed on assertions.

4 Discussion Topics

A large part of the afternoon sessions of the workshop was devoted to group discussions. Four main discussion topics were discussed. These group discussions are summarized below.

4.1 Discussion Group 1: Aspects, Dependencies and Interactions Due to and/or Prohibited by Languages

The goal of this discussion group was to investigate the role of programming language design as a cause or a means to avoid unwanted interactions. The idea behind this topic was that (a) the features of programming languages, such as aspects, can be the enabler for certain interactions, both desired and undesired, and (b) hence there are trade-offs to be made such as expressiveness versus interactions. Some examples are listed next:

- If there were no join points within advice code, there would be no undesired infinite loops caused by advice that is called while executing itself.
- Allowing for pointcuts or advice to ignore the regular OO encapsulation rules, creates potential for unwanted dependencies between aspects and base code.
- The ability to affect the control flow of the base code (e.g., by omitting a *proceed()* statement within *around* advice) is powerful, but can also easily destroy the correctness of the application.

One of the first issues that was discussed was whether or not a language should allow to specify interactions, and/or avoid conflicts. Avoiding conflicts in general is very hard, though, without severely reducing the expressiveness of the language. Hence, the group considered that additional specifications would be necessary to allow for the static detection (and hence avoidance) of interference: for example in the form of contracts or relation specifications.

It was questioned by Shigeru Chiba that many new features for AOP are proposed, often without convincing cases to motivate them, and raising the question whether these features are not merely useful for certain applications or application domains only. In particular, he suggested that many examples of aspects were the codification

of program idioms, where there might be other, more conventional ways, such as mix-ins, generics or C++ templates, to express the same behavior.

Further, the group discussed the kind of interferences that are caused by AOP. First, it was proposed that these must be strongly related to the identifying properties AOP, such as obliviousness and quantification. However, it was concluded that in fact, AOP does not introduce new types of interference, but only makes it easier to create them. The reason is that AOP offers new and more powerful composition mechanisms, but in the end, these result in the same types of behavioral combination that can be created manually in procedural or object-oriented languages.

4.2 Discussion Group 2: Aspects, Dependencies and Interactions in Applications

People that joined this discussion group came from two different backgrounds: software product lines and middleware services. Discussion started by agreeing on the fact that a feature in the product line world matches a service in a middleware context. In addition, all discussion participants believed that the true power of AOSD lies in quantification (i.e., composition) rather than in obliviousness. One of the main problems with complete obliviousness is that the aspect developer needs to be aware of the entire system. We refer the reader for the remainder of this discussion topic to Section 5.

Second topic within this discussion group was crosscutting programming interfaces, XPI's, work from Griswold et al. [10] It was concluded that these XPI's would be a very nice idea especially in the context of feature development because a more safe evolution and composition becomes possible. The discussants highlighted the issue of defining such a crosscutting programming interface. At first sight, the base code developer seems more appropriate to define this interface because he/she knows the code the best. But it's hard to imagine that the base code developer knows of all other pieces of code that will cooperate with the base code. Obviously, one of the real problems in specifying pointcuts is that they are mainly syntax-based, which gives rise to the fragile pointcut problem. The group concluded that domain-specific aspect languages might be used as a source of inspiration when trying to raise the level of abstraction.

The next discussion topic in this group consisted of the notion of two-sided contracts as in [16]. The group considered this to be a very interesting idea. The idea of aspect categories and explicitly stating which categories are allowed at a certain point seems a useful thing to do. In addition, the approach enables manageability and safety at the same time. However, the question was posed if the contracts as proposed in [16] are expressive enough.

The group ended this discussion session by reflecting on what should be expressed? On the one hand, expressing what is allowed depends on the specific requirements within a particular application context while expressing what is not allowed seems to pose difficulties taking evolution into account.

4.3 Discussion Group 3: State and Future of Formal Methods for Aspects

A lively discussion on the state and future for formal methods focused on two main questions:

1. What kind of properties are of particular interest for AOSD?
2. What methods can be used to analyze and ensure such properties?

Aspect interactions were discussed as a prime example of properties relevant to AOSD. The current notion of interactions between aspects that are applied to the same joinpoint was identified as a major stumbling block for the handling of interactions among aspects. This coarse notion of interactions forbids the analysis of the frequent case where interactions are caused by two aspects manipulating a common state at different joinpoints. As a second group of aspect-relevant properties, security properties at different levels of abstraction (e.g., on the heap level, on the level of calls to higher-level services) have been discussed.

The discussion on formal methods for the analysis and verification of aspect-relevant properties centered on the need for easy to use, robust and scalable tools. Currently, almost no existing tool supports more than one of those criteria. A major underlying cause for this state of affairs is a lack of modular analysis and verification methods for aspects: specifications of formal properties therefore are rather unwieldy and require time-consuming whole-program analyses. Approaches that restrict aspects on the basis of traditional module boundaries as well as pre-computation of analysis information for program parts that can be reused in the context of analyses on larger programs were discussed as potential solutions to these problems.

4.4 Discussion Group 4: A Classification of Aspect Dependencies and Interactions

The topic of this discussion group was classifications for aspect dependencies and interactions. As it was clear that the group would not be able to provide such a classification or a classification framework, it looked into the characteristics of classifications for aspect dependencies and interactions.

The discussion started from the following question: "Why are there so many classifications for aspect dependencies and interactions?" By this question, the group referred to the situation that in ADI there are relatively many different classifications given that it is still a small community and compared to the overall number of papers on ADI. Some of the noteworthy classifications are [17, 5, 7] but this list is by no means complete.

Firstly, the group agreed on the fact that classifications are useful because they are a means for understanding the problem space, i.e., the possible dependencies and interactions among aspects or among aspects and other kind of modules. Classifications are also a means to classify the solution space, i.e., the approaches to detect or handle such dependencies. Classifications help with building tools and allow comparing different approaches and tools. A useful classification should cover commonalities and variabilities. Lastly, a useful classification should have been successfully used more than once.

Classifications for aspect dependencies and interactions differ in the following dimensions. These dimensions apply to the problem space that is introduced above. The group considered these dimensions as orthogonal to each other.

- Development phases of the software development life cycle, i.e., requirements engineering, architecture, design modeling, implementation, and testing.
- Expressive power of aspect languages.

– Level of abstraction.

It equally makes sense to think of classifications for the solution space, i.e., for the approaches that detect or even solve aspect dependencies and interactions. These approaches will use theoretical foundations. Therefore, the solution space can be classified according to the complexity and limitations of the theoretical foundations used.

Getting back to the question that the group asked itsef at the beginning of the discussion, our conclusion is that there is potential for harmonization and unification of classifications.

5 Panel on "Does AO Equal Quantification and Obliviousness?"

The workshop hosted a panel that discussed the question "Does AO equal quantification and obliviousness?" [8]. In particular, the panelists had to formulate an answer to the following three questions.

1. Are quantification and obliviousness fundamental to AO?
2. If yes, why should we embrace them?
3. If no, then what is AO about?

We first elaborate on the different panel positions in which each of the four panelists presented his personal view on the matter. Next, an overview is given from the panel discussion based on questions from the workshop attendants.

5.1 Panel Positions

Michael Haupt started by declaring that we shouldn't be dogmatic: AOSD is about getting some constructs to modularize crosscutting concerns. There are some concerns we can modularize with OO, others we can't. For these, we introduced the term aspects and that's also what AO should be about: modularizing those crosscutting concerns. W.r.t. obliviousness, concerns are already there even before any code is being written. They are an inherent part of a system instead of imposed on (part of) a system. In addition, crosscutting concerns are crosscutting by nature: we can't do anything about it. Obliviousness, however, means in its original definition that aspects can just be imposed on (parts of) a system. The analogy with patches was thrown, where modules don't know they are being patched similar to modules that are oblivious to the fact if there is an aspects imposed on them. This clearly results in a contradiction with crosscutting concerns being there from the start. Michael concluded his statement on obliviousness with requiring that any part of the system should not be more obliviousness to any other part than in traditional OO. For the quantification part of the questions, a very important question in his opinion regards what we should quantify over? Nowadays, we are able to quantify over both static and dynamic parts of a system. We definitely must not quantify over internals of modules, but over interfaces. This way, modules are allowed to express themselves in terms of situations that may be of interest to other modules without giving away too many details.

Klaus Ostermann doesn't like the word obliviousness much because it refers to code locations and as a result, an aspect refers to a module. But an aspect affects

a point in the dynamic flow of a program and not a module. Aspects should offer a modular implementation of global invariants of the form: "whenever X happens, do Y". Otherwise, it is implied that we only can understand programs in a step-by-step manner rather than having some higher level of understanding, while the latter is exactly what we should aim for. Many examples have proven this to be true: thread yielding, garbage collection, lazy evaluation, email filtering, etc. A major problem in AOSD so far is that one aspect may destroy the higher-level invariant that is assumed by another aspect. Therefore, in future work, more attention should go to the more controlled interaction between the invariants, in such a way that the problem is composed of modules where each module is responsible for maintaining one or more invariants.

Hidehiko Masuhara rephrased the title of the panel slightly to "AO = quantification (+ obliviousness not necessary) + join point abstraction" because the latter is often overlooked. AO mechanisms can be seen as means of identifying join points and affecting the behavior at those join points, in parallel with the 3-part model that Masuhara et al. have proposed at ECOOP 2003 [15]. When comparing both pieces of work, quantification nicely matches with the means of identifying join points. On the other hand, obliviousness more or less equals how the means of identifying and affecting are modeled, which is not an essential part of the 3-part model. However, join point abstraction, which is not explicitly mentioned in both models, should enable us to capture multiple join points at once. This is often supported by giving a name to a set of join points, so the details are hidden from the user. One of the other panelists asked the speaker about the difference between quantification and join point abstraction. This was countered by explaining that both are not the same thing. The speaker ended with the open question "Is naming sufficient to provide abstraction?"

Wouter Joosen sketched the following historical perspective. When we moved from procedural to OO programming, we went for localization. In this regard, encapsulation can be considered as a first wave of modularization. When we moved to aspects, this only happened because the modularization in OO was not enough: crosscutting concerns still existed. But when going to a next, extended, paradigm, we should not throw away OO ideas. Transactions, persistence and security are the three reusable services that one wants to configure without re-implementing everything over and over again. And, (un)fortunately, obliviousness here exactly is the crime for AO, such as for instance motivated by [22, 12]. Due to the current context, we should choose another term (suggestion: dependencies) and take it from there. Finally, some observations were given to the audience. Firstly, the time to ship a software product is essential and makes shifting to components necessary. Aspect should be combined with components if they want to be useful in production environments. Secondly, quantification needs to be over interfaces. Last, but not least, the idea to document the effect of advice next to the effect of aspects [5] is a very valuable one.

5.2 Panel Discussion

Discussions were centered around three more specific topics: obliviousness, interfaces and abstraction. Summaries for each of these discussions are provided below.

Obliviousness At the beginning of the discussion, the workshop participants agreed that the developer of a module best knows what the module can expose and what not. This should not be influenced by aspects. This is exactly what was pointed out

before by some of the panelists. Hidehiko Masuhara complemented this line of thinking, which is similar to Aldrich's open modules [1], by pointing out that the abstraction itself is important, no matter who defines the abstraction where. Hidehiko Masuhara reminded us of the meta-level programming world of computational reflection and asked the question "How do we distinguish AO from meta-level programming?". Klaus Ostermann pointed out that meta-level programming goes about a program with its syntax while AO, at least, tries to talk about the semantics of a program.

According to the panelists, *obliviousness* seems to be a negative thing. Is there any idea about how to design AO technologies without obliviousness? Given the fact that AspectJ has built in some property of obliviousness, is it possible to take it away? In other words, are quantification and obliviousness truly essential? For obliviousness, the answer would be that it is not essential, but useful. Obviously, not all AO technologies must have obliviousness mechanisms. One example would be using AspectJ only with pointcuts on annotations. As a consequence, obliviousness is not essential, since AO technologies can do without it. Languages of course also carry more or less obliviousness than others with different degrees of coupling and cohesion. Klaus Ostermann complemented this with stating that, in his opinion, a pointcut can involve private methods as long as the implementation details are kept hidden. To illustrate, if an aspect depends on the name of a method in AspectJ, one would interpret this as anti-modular because changing implementation details can invalidate aspects. However, pointcuts can either be formulated by referring to method names or by using higher level pointcuts. The latter clearly does not break encapsulation.

Interfaces A member of the audience also pointed out that the revised definitions of quantification and obliviousness have been overlooked in the discussion so far. Wouter Joosen responded that obliviousness stays a crime and in essence is all about dependencies. Looking at the base code, do we have to know there are dependencies? XPI's [10] and Open Modules [1] are about specifying points that you cannot see in code. We certainly do not want annotations everywhere and a limited form of obliviousness sounds appealing. In any case, we need a mechanism to express what we expect. If you say in an AO composition that you imposed behaviour, then there should be contracts that say which compositions are allowed or not. We can be more precise about what we may want in terms of aspect composition. But if we don't want to be that precise, then we should not violate existing contracts and break encapsulation of, e.g., private elements. Michael Haupt illustrated this further by indicating that there exist different ways in AspectJ to interact with a module ending up with something not being a module any longer if encapsulation gets broken.

An audience member acknowledged Michael Haupt's suggestion that we should plan ahead. However, he claimed that some crosscutting concerns arise because of requirements changes, often after the code is in place. Aspects were compared to patches in this regard. Immediately, the panel intervened by declaring we have a choice. For instance, if there is something in an OO system that doesn't fit my needs, I ask the developer of the module to create a new abstraction. The audience questioned how that developer then could create that new abstraction? One would think using obliviousness, probably. Michael Haupt responded that another option is to have the ability in future AO languages for a module to expose its own interface, and not have an external entity responsible for this. Since, the base module developer has full control over the code, he is the best person in place to provide such an abstraction. Another member of

the audience raised the concern that you have component programming on the one side and component assembly on the other. At the assembly-level, you absolutely need contracts in a non-oblivious manner. Wouter Joosen complemented this with noting that oblivousness is of little value when the programmer has full control over the code. The latter is very important from the perspective of software industry where things need to be shipped that won't break. Everybody agreed on the following conclusion of this discussion topic. We should differentiate between obliviousness w.r.t. interfaces and obliviousness w.r.t. implementation. We should avoid the former while the latter is acceptable.

Abstraction On a question what makes AO different, Hidehiko Masuhara answered with join point abstraction. Klaus Ostermann's view on this matter regards the ability to declaratively describe interesting events and to use this mechanism to implement invariants. Among others, the parallel with macros was drawn. The position of the panel was that an aspect is exposed as a first-class construct dynamically while a macro is done completely statically. The power of using *proceed()* to manipulate the control flow of the program dynamically is not available when working with macros. At the moment, aspects seem to be a sort of swiss army knife doing everything from replacing a byte code rewriter to implementing different crosscutting concerns. This highlights the importance to recognise different needs for obliviousness instead of discouraging obliviousness all together.

The audience started another discussion from the viewpoint of long term maintenance of a product. What happens if changes are required in the next release of a product? If in the previous release the base programmers weren't aware, then in this next release they should be. The relevance is clear if changes to the base code are needed of which the base programmers know that it would affect the aspects that rely on it. Wouter also pointed out the relevance of verifying any interaction that is modeled in a contract, such as for instance in [9], by the compiler or a dedicated verifier. This relates to the ideas in [19].

Finally, Awais Rashid challenged the panel asking what abstraction is essentially? Is naming a sufficient property for abstraction? Hidehiko Masuhara acknowledged the fact that, nowadays, the only property we have at our disposal is naming. Klaus Ostermann completed the position of the panel by stating that naming alone probably will not be enough. Everyone agreed that join point abstraction will be one of the key differentiating factors. A join point definition is abstract if and only if it is in terms of the domain at hand rather than a projection to the code. An audience member asked if it was realistic to try to abstract the pointcut specifications from the code and turn that into an interface and then find an aspect that matches that interface? Would this be sufficient? Hidehiko Masuhara emphasized the resemblance with XPI's [10]. Whether the AspectJ pointcut language suffices to achieve this is another question. Klaus Ostermann also referred to a paper of Kiczales et al. about aspect-aware interfaces [11] at ICSE 2005.

6 Conclusion

This second workshop on Aspects, Dependencies and Interactions provided an opportunity for presentations and lively discussion between researchers working on AOSD,

dependencies and interactions from all over the world. The workshop continued the wide discussion on aspects, dependencies and interactions that was started at last years' ADI 2006. It is our intention to continue encouraging the challenging work on this topic by further organizing a number of follow-up workshops.

7 Workshop Organizers and Participants

7.1 List of Organizers

The workshop organizing committee consisted of the following five members.

- Frans Sanen, K.U.Leuven, Belgium (co-chair)
  Email: frans.sanen (at) cs.kuleuven.be
- Ruzanna Chitchyan, Lancaster University, UK (co-chair)
  Email: rouza (at) comp.lancs.ac.uk
- Lodewijk Bergmans, University of Twente, The Netherlands
  Email: L.M.J.Bergmans (at) ewi.utwente.nl
- Johan Fabry, INRIA Futurs, France
  Email: johan.fabry (at) lifl.fr
- Mario Sudholt, Ecole des Mines de Nantes, France
  Email: mario.sudholt (at) emn.fr

7.2 List of Attendees

The list of attendees officially registered for the workshop is presented alphabetically below. It should be noted that a number of unregistered attendees also participated, but these are not listed here.

1. Zaid Altahat (Illinois Institute of Technology, USA)
2. Mourad Badri (Universit du Qubec Trois-Rivires, Canada)
3. David Bar-On (Open University of Israel, Israel)
4. Jorge Barreiros (Instituto Politecnico de Coimbra, Portugal)
5. Benoit Baudry (IRISA, France)
6. Alexandre Bergel (University of Potsdam, Germany)
7. Lodewijk Bergmans (University of Twente, The Netherlands)
8. Julien Charles (INRIA, France)
9. Shigeru Chiba (Tokyo Institute of Technology, Japan)
10. Johan Fabry (INRIA Futurs, France)
11. Gael Fraeteur (PostSharp, Czech Republic)
12. Birgit Grammel (SAP AG, Germany)
13. Phil Greenwood (Lancaster University, UK)
14. Florian Heidenreich (Dresden University of Technology, Germany)
15. Kevin Hoffman (Purdue University, USA)
16. Atsushi Igarashi (Kyoto University, Japan)
17. Jendrik Johannes (Dresden University of Technology, Germany)
18. Wouter Joosen (K.U.Leuven, Belgium)
19. Bert Lagaisse (K.U.Leuven, Belgium)
20. Gary T. Leavens (IOWA State University, USA)
21. Hidehiko Masuhara (University of Tokyo, Japan)

22. Katharina Mehner (Siemens, Germany)
23. Klaus Ostermann (Technical University of Darmstadt, Germany)
24. Meir Ovadia (Cadence Design Systems, USA)
25. Marco Piccioni (ETH Zurich, Switserland)
26. Awais Rashid (Lancaster University, UK)
27. Frans Sanen (K.U.Leuven, Belgium)
28. Hans Schippers (University of Antwerp, Belgium)
29. Sergio Soares (Universidade de Pernambuco, Brazil)
30. Guido Soldner (FAU Erlangen, Germany)
31. Fredrik Sorensen (University of Oslo, Norway)
32. Mario Sudholt (Ecole des Mines de Nantes, France)
33. Shmuel Tyszberowicz (Open University of Israel, Israel)
34. Naoyasu Ubayashi (Kyushu Institute of Technology, Japan)

References

1. J. Aldrich. Open modules: Modular reasoning about advice. In *Proceedings of the European Conference on Object-Oriented Programming, LNCS 3586*, pages 144–168, 2005.
2. Aspectj. http://www.eclipse.org/aspectj.
3. D. Bar-On and S. Tyszberowicz. Derived requirements generation. In *Proceedings of the Second International Workshop on Aspects, Dependencies and Interactions (held at ECOOP)*, pages 5–10, 2007.
4. J. Barreiros and A. Moreira. Aspect interaction management with meta-aspects and advice cardinality. In *Proceedings of the Second International Workshop on Aspects, Dependencies and Interactions (held at ECOOP)*, pages 11–16, 2007.
5. C. Clifton and G. T. Leavens. Spectators and assistants: Enabling modular aspect-oriented reasoning. Technical Report TR02-10, Iowa State University, 2002.
6. D. S. Dantas and D. Walker. Harmless advice. *33rd ACM SIGPLAN - SICACT Symposium on Principles of Programming Languages (POPL06)*, 41(1):383–396, 2006.
7. R. Douence, P. Fradet, and M. Südholt. Composition, reuse, and interaction analysis of stateful aspects. In *Proceedings of the 3rd international Conference of Aspect-oriented Software Development*, Lancaster, UK, 2004. ACM.
8. R. Filman and D. Friedman. Aspect-oriented programming is quantification and obliviousness. In *Proceendings of Workshop on Advanced Separation of Concerns, OOPSLA 2000*, 2000. October 2000, Minneapolis. http://ic-www.arc.nasa.gov/ic/darwin/oif/leo/filman/text/oif/aop-is.pdf.
9. P. Greenwood, G. Coulson, A. Rashid, B. Lagaisse, F. Sanen, E. Truyen, and W. Joosen. Interactions in aspect-oriented middleware. In *Proceedings of the Second International Workshop on Aspects, Dependencies and Interactions (held at ECOOP)*, pages 17–22, 2007.
10. W. G. Griswold, K. Sullivan, Y. Song, M. Shonle, N. Tewari, Y. Cai, and H. Rajan. Modular software design with crosscutting interfaces. *IEEE Software*, 23(1):51–60, 2006.
11. G. Kiczales and M. Mezini. Aspect-oriented programming and modular reasoning. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 49–58, New York, NY, USA, 2005. ACM Press.
12. J. Kienzle and S. Gélineau. Ao challenge - implementing the acid properties for transactional objects. In *Proceedings of the 5th International Conference on Aspect-Oriented Software Development*, pages 202–213, New York, NY, USA, 2006. ACM Press.
13. S. Kojarski and D. H. Lorenz. Awesome: A co-weaving system for multiple aspect-oriented extensions. In *Proceedings of the 22nd Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, ACM Press, 2007.

14. D. H. Lorenz and S. Kojarski. Understanding aspect interactions, co-advising and foreign advising. In *Proceedings of the Second International Workshop on Aspects, Dependencies and Interactions (held at ECOOP)*, pages 23–28, 2007.
15. H. Masuhara and G. Kiczales. Modeling crosscutting in aspect-oriented mechanisms. In *Proceedings of European Conference on Object-Oriented Programming, LNCS 2743*, pages 2–28, 2003.
16. F. Munoz, O. Barais, and B. Baudry. Vigilant usage of aspects. In *Proceedings of the Second International Workshop on Aspects, Dependencies and Interactions (held at ECOOP)*, pages 29–35, 2007.
17. M. Rinard, A. Sălcianu, and S. Bugrara. A classification system and analysis for aspect-oriented programs. In *Proceedings of SIGSOFT'04/FSE-12*, pages 147–158, Newport Beach, CA, USA, 2004. ACM.
18. É. Tanter and J. Noyé. A versatile kernel for multi-language AOP. In R. Glck and M. Lowry, editors, *Proceedings of the 4th ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE 2005)*, volume 3676, pages 173–188, Tallinn, Estonia, 2005.
19. N. Ubayashi, Y. Maeno, K. Noda, and G. Otsubo. A verification mechanism for weaving in extensible aom languages. In *Proceedings of the Second International Workshop on Aspects, Dependencies and Interactions (held at ECOOP)*, pages 36–41, 2007.
20. N. Ubayashi, T. Tamai, S. Sano, Y. Maeno, and S. Murakami. Model compiler construction based on aspect-oriented mechanisms. In R. Glck and M. R. Lowry, editors, *GPCE*, volume 3676 of *Lecture Notes in Computer Science*, pages 109–124. Springer, 2005.
21. N. Ubayashi, T. Tamai, S. Sano, Y. Maeno, and S. Murakami. Aspect-oriented and collaborative systems metamodel access protocols for extensible aspect-oriented modeling. In K. Zhang, G. Spanoudakis, and G. Visaggio, editors, *SEKE*, pages 4–10, 2006.
22. B. D. Win. Engineering application-level security through aspect-oriented software development. PhD dissertation, 2004.